View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Universidade do Minho: RepositoriUM

Coinductive Interpreters for Process Calculi

L. S. Barbosa and J. N. Oliveira

Departamento de Informática, Universidade do Minho, Braga, Portugal
{lsb,jno}@di.uminho.pt

Abstract. This paper suggests *functional programming* languages with coinductive types as suitable devices for prototyping *process calculi*. The proposed approach is independent of any particular process calculus and makes explicit the different ingredients present in the design of any such calculi. In particular structural aspects of the underlying *behaviour model* (*e.g.*, the dichotomies such as active *vs* reactive, deterministic *vs* non-deterministic) become clearly separated from the *interaction* structure which defines the synchronisation discipline. The approach is illustrated by the detailed development in Charity of an interpreter for a family of process languages.

Keywords: functional programming, applications, coinductive types.

1 Introduction

The essence of *concurrent* computation lies in the fact that a transition in a system may *interact* with, or depend upon, another transition occurring in a different system executing alongside. Since this was observed by C. A. Petri [23], in the early sixties, several process calculi have been proposed to specify and reason about concurrent and communicating systems. The diversity of semantic models proposed (emphasising *e.g.*, linear or branching temporal structures, causality or interleaving, synchrony or asynchrony) witness both the difficulty of understanding concurrency and the practical relevance of the topic ([27] provides a comprehensive survey).

This paper argues that *declarative programming*, in its *functional* flavour, may provide a suitable vehicle for prototyping such calculi and assess alternative design decisions. Such a role for functional languages has long been established in the formal methods community, at least since P. Henderson's me too [10] proposal for animating Vdm [15] specifications. However, only static, data-oriented, aspects of computational systems are usually considered there.

Our starting point is the well known fact that *initial* algebras and *final* coalgebras, underlying *inductive* and *coinductive* types, provide abstract descriptions of a variety of phenomena in programming. In particular, of *data* and *behavioural* structures, respectively [5, 12].

Initiality and finality, as (categorical) universal properties, entail definitional as well as proof principles, *i.e.*, a basis for the development of program calculi

directly based on (*i.e.*, actually driven by) type specifications. Moreover, such properties can be turned into programming *combinators* which are parametric on the *shape* of the type structure, captured by a signature of operations encoded in a suitable functor. These can be used, not only to reason about programs, but also to program with. In fact, such combinators have been incorporated on real programming languages as *polytypic* functionals, generalising the well-known `map`, `fold` and `unfold` constructs (see, *e.g.*, [25, 14, 13]).

In this paper we resort to the experimental programming language CHARITY [6] to prototype process calculi and quickly implement simple interpreters for the corresponding process languages. As a programming language, CHARITY is based on the term logic of distributive categories and provides a definitional mechanism for categorical datatypes *i.e.*, for both initial algebras and final coalgebras. Processes can therefore be directly represented as inhabitants of coinductive types, *i.e.*, of the carriers of final coalgebras for suitable Set endofunctors.

We believe this approach has a number of advantages:

- First of all, it provides a *uniform* treatment of processes and other computational structures, *e.g.*, data structures, both represented as categorical types for functors capturing signatures of, respectively, observers and constructors. Placing data and behaviour at a similar level conveys the idea that process models can be chosen and specified according to a given application area, in the same way that a suitable data structure is defined to meet a particular engineering problem. As a prototyping platform, CHARITY is a sober tool in which different process combinators can be defined, interpreters for the associated languages quickly developed and the expressiveness of different calculi compared with respect to the intended applications. We believe that prototyping has an important role in any specification method, as it supports stepwise development. In particular, it allows each design stage to be immediately animated and quick feedback about its behaviour gathered.
- The approach is independent of any particular process calculus and makes explicit the different ingredients present in the design of any such calculi. In particular structural aspects of the underlying *behaviour model* (*e.g.*, the dichotomies such as active *vs* reactive, deterministic *vs* non-deterministic) become clearly separated from the *interaction* structure which defines the synchronisation discipline.
- Finally, as discussed in an author's previous paper [3], proofs of process properties can be done in a calculational (basically *equational* and *pointfree*) style, therefore circumventing the explicit construction of bisimulations typical of most of the literature on process calculi[1].

2 Preliminaries

It is well known that the signature of a data type, *i.e.*, its contractual interface, may be captured by a *functor*, say T, and that a canonical representative of

[1] This sort of *proofs by calculation* is addressed in [3]; see also [2].

the envisaged structure arises as a solution, *i.e.*, as fixed point, of the equation $X \cong \mathsf{T}\ X$. In fact, an abstract data structure is defined as a T-*algebra*, *i.e.*, a map $t : \mathsf{T}\ D \longrightarrow D$ which specifies how values of D are built using a collection of constructors, recorded in T. The canonical representative of such T-algebras is the (initial) term algebra.

There are, however, several phenomena in computing which are hardly definable (or even simply not definable) in terms of a complete set of constructors. This leads to *coalgebra* theory [24]. While in algebra data entities are built by constructors and regarded as different if 'differently constructed', coalgebras deal with entities that are observed, or decomposed, by observers and whose internal configurations are identified if not distinguishable by observation. Given an endofunctor T, a T-coalgebra is simply a map $p : U \longrightarrow \mathsf{T}\ U$ which may be thought of as a transition structure, of *shape* T, on a set U, usually referred to as the *carrier* or the *state space*. The shape of T reflects the way the state is (partially) accessed, through *observers*, and how it evolves, through *actions*. As a consequence, equality has to be replaced by *bisimilarity* (*i.e.*, equality with respect to the observation structure provided by T) and coinduction replaces induction as a proof principle. For a given T, the final coalgebra consists of all possible behaviours up to bisimilarity, in the same sense that an initial algebra collects all terms up to isomorphism. This is also a fixpoint of a functor equation and provides a suitable universe for reasoning about behavioural issues. This has lead to some terminology for final coalgebras: *coinductive* or *left datatypes* in [9] or [7], *codata* and *codatatypes* [16], *final systems* in [24] or *object types* in [11].

A T-coalgebra *morphism* between two T-coalgebras, p and q, is a map h between their carriers making the following diagram to commute

$$\begin{array}{ccc} U & \xrightarrow{p} & \mathsf{T}\ U \\ {\scriptstyle h}\downarrow & & \downarrow{\scriptstyle \mathsf{T}\ h} \\ V & \xrightarrow{q} & \mathsf{T}\ V \end{array}$$

The *unique* such morphism to the final coalgebra $\omega_{\mathsf{T}} : \nu_{\mathsf{T}} \longrightarrow \mathsf{T}\ \nu_{\mathsf{T}}$ from any other coalgebra $\langle U, p\rangle$ is called a T-*anamorphism* [20], or the *coinductive extension* of p [26]. It is written $[\![(p)]\!]_{\mathsf{T}}$ or, simply, $[\![(p)]\!]$, and satisfies the following universal property,

$$k = [\![(p)]\!]_{\mathsf{T}} \quad \Leftrightarrow \quad \omega_{\mathsf{T}} \cdot k = \mathsf{T}\ k \cdot p$$

being unique up to isomorphism.

Its dual, in the algebraic side, is the unique arrow $(\![d]\!)_{\mathsf{T}}$, or simply $(\![d]\!)$, from the initial T-algebra to any other algebra d, known as a *catamorphism*. Both d here and p above are referred in the sequel as the catamorphism (resp., anamorphism) *gene*.

Charity, the prototyping language used in this paper, provides direct support for both initial algebras and final coalgebras for strong functors. The *strong* qualification means that the underlying functor T possess a *strength* [17, 7], *i.e.*,

a natural transformation $\tau_r^{\mathsf{T}} : \mathsf{T} \times - \Longrightarrow \mathsf{T}(\mathsf{Id} \times -)$ subject to certain conditions. Its effect is to *distribute* context along functor T. When types are modeled in such a setting, the universal combinators (as, *e.g.*, cata and anamorphisms) will possess a somewhat more general shape, able to deal with the presence of extra parameters in the functions being defined. This holds, of course, even when the underlying category is not cartesian closed (and therefore *currying* is not available).

3 Processes and Combinators

3.1 Processes

The operational semantics of a process calculus is usually given in terms of a transition relation $\xrightarrow{a}$ over processes, indexed by a set *Act* of actions, in which a process gets committed, and the resulting 'continuations', *i.e.*, the behaviours subsequently exhibited. A first, basic design decision concerns the definition of what should be understood by such a collection. As a rule, it is defined as a *set*, in order to express non-determinism. Other, more restrictive, possibilities consider a sequence or even just a single continuation, modelling, respectively, 'ordered' non-determinism or determinism. In general, this underlying behaviour model can be represented by a functor B.

An orthogonal decision concerns the intended interpretation of the transition relation, which is usually left implicit or underspecified in process calculi. We may, however, distinguish between

- An 'active' interpretation, in which a transition $p \xrightarrow{a} q$ is informally characterised as 'p evolves to q by performing an action a', both q and a being solely determined by p.
- A 'reactive' interpretation, informally reading 'p reacts to an external stimulus a by evolving to q'.

Processes will then be taken as inhabitants of the carrier of the final coalgebra $\omega : \nu \longrightarrow \mathsf{T}\ \nu$, with T defined as $\mathsf{B}\ (Act \times \mathsf{Id})$, in the first case, and $(\mathsf{B}\ \mathsf{Id})^{Act}$, in the second. To illustrate our approach, we shall focus on the particular case where B is the finite powerset functor and the 'active' interpretation is adopted. The transition relation, for this case, is given by $p \xrightarrow{a} q \quad \mathsf{iff} \quad \langle a, q\rangle \in \omega\ p$.

The restriction to the finite powerset avoids cardinality problems and assures the existence in Set of a final coalgebra for T. This restricts us to *image-finite* processes, a not too severe restriction in practice which may be partially circumvented by a suitable definition of the structure of *Act*. For instance, by taking *Act* as *channel* names through which data flows. This corresponds closely to 'CCS with value passing' [21]. Therefore, only the set of channels, and not the messages (seen as pairs channel/data), must remain finite. In fact, as detailed below, an algebraic structure should be imposed upon the set *Act* of actions in order to capture different interaction disciplines. This will be called an *interaction structure* in the sequel.

Down to the prototype level, we will start by declaring a process space as the coinductive type `Pr(A)`, parametrized by a specification `A` of the interaction structure:

```
data C -> Pr(A) = bh: C -> set(A * C).
```

where `set` stands for a suitable implementation of (finite) sets. See appendix B for details about the syntax of CHARITY.

3.2 Interaction

We first assume that actions are generated from a set L of *labels*, *i.e.*, a set of formal names. Then, an *interaction structure* is defined as an Abelian positive monoid $\langle Act; \theta, 1\rangle$ with a zero element 0. It is assumed that neither 0 nor 1 belong to the set L of labels. The intuition is that θ determines the interaction discipline whereas 0 represents the absence of interaction: for all $a \in Act$, $a\theta 0 = 0$. On the other hand, the monoid being being positive implies $a\theta a' = 1$ iff $a = a' = 1$. Notice that the role of both 0 and 1 is essentially technical in the description of the interaction discipline. In some situations, 1 may be seen as an *idle* action, but its role, in the general case, is to equip the behaviour functor with a *monadic* structure, which would not be the case if *Act* were defined simply as an Abelian semigroup.

Let us consider two examples of interaction structures. For each process calculus, actions over L are introduced as an inductive type `Ac(L)` upon which an equality function and a product θ are defined.

Co-occurrence. A basic example of an interaction structure captures action co-occurrence: θ is defined as $a\theta b = \langle a, b\rangle$, for all $a, b \in Act$ different from 0 and 1. The corresponding type, parametric on the set `L` labels, is therefore defined inductively as follows

```
data Ac(L) -> A =
     act: L -> A | syn: A * A -> A | nop: 1 -> A | idle: 1 -> A.
```

The embedding of labels into actions is explicitly represented by the constructor `act`. The distinguished actions 0 and 1 are denoted by `nop` and `idle`, respectively. Action co-occurrence, for any actions `a` and `b` different from 0 and 1, is represented by `syn(a,b)`. The specification is complete with a definition of action product θ, encoded here as the function `prodAc`, and an equality predicate `eqA` on actions, both parametric on `L`. The actual CHARITY code for θ is as follows,

```
def prodAc : Ac(L) * Ac(L) -> Ac(L)
    = (nop, _ ) => nop  | ( _ ,nop) => nop | (idle, x) => x
    | (x ,idle) => x    | ( a1, a2) => syn( a1, a2).
```

The CCS case. CCS [21] synchronisation discipline provides another example. In this case the set L of labels carries an involutive operation represented by an horizontal bar as in $\overline{a}$, for $a \in L$. Any two actions a and $\overline{a}$ are called complementary. A special action $\tau \notin L$ is introduced to represent the result of a synchronisation between a pair of complementary actions. Therefore, the result of θ is τ whenever applied to a pair of complementary actions and 0 in all other cases, except, obviously, if one of the arguments is 1.

We first introduce the involutive complement operation on labels by replacing `L` by the parametric type `Lb(N)` and defining a label equality function `eqL`.

```
data Lb(N) -> I = name: N -> I | inv: I -> I.
```

Then the interaction structure is defined by `Ac(L)` as above, together with action equality and product:

```
def prodAc{eqL: Lb(N) * Lb(N) -> bool} : Ac(Lb(N)) * Ac(Lb(N)) ->  Ac(Lb(N))
    = (act(1),act(k)) |  or(eqL(l,inv(k)),eqL(inv(l),k) => tau
                      |  ..  => nop
    | (idle, x) => x  |  (x ,idle) => x  |   _       => nop.
```

3.3 Dynamic Combinators

The usual CCS dynamic combinators — *i.e.*, *inaction*, *prefix* and non-deterministic *choice* — are defined as operators on the final universe of processes. Being no recursive, they have a direct (coinductive) definition which depends solely on the chosen process structure. Therefore, the *inactive process* is represented as a constant $\mathsf{nil} : \mathbf{1} \longrightarrow \nu$ upon which no relevant observation can be made. *Prefix* gives rise to an *Act*-indexed family of operators $a. : \nu \longrightarrow \nu$, with $a \in Act$. Finally, the possible actions of the *non-deterministic choice* of two processes p and q corresponds to the collection of all actions allowed for p and q. Therefore, the operator $+ : \nu \times \nu \longrightarrow \nu$ can only be defined over a process structure in which observations form a collection. Formally,

$$
\begin{array}{rrcl}
\text{inaction} & \omega \cdot \mathsf{nil} & = & \underline{\emptyset} \\
\text{choice} & \omega \cdot + & = & \cup \cdot (\omega \times \omega) \\
\text{prefix} & \omega \cdot a. & = & \mathsf{sing} \cdot \mathsf{label}_a
\end{array}
$$

where $\mathsf{sing} = \lambda x\,.\ \{x\}$ and $\mathsf{label}_a = \lambda x\,.\ \langle a, x\rangle$. These definitions are directly translated to CHARITY as functions `bnil`, `bpre` and `bcho`, respectively:

```
def bnil: 1 -> Pr(A)
    = () => (bh: empty).

def bpre: A * Pr(A) -> Pr(A)
    = (a, t)  => (bh: sing(a,t)).

def bcho: Pr(A) * Pr(A)  -> Pr(A)
    = (t1, t2) => (bh: union(bh t1, bh t2)).
```

3.4 Static Combinators

Persistence through action occurrence leads to the recursive definition of *static* combinators. This means they arise as anamorphisms generated by suitable 'gene' coalgebras. *Interleaving*, *restriction* and *renaming* are examples of *static* combinators, which, moreover, depend only on the process structure. On the other hand, *synchronous product* and *parallel composition* also rely on the interaction structure underlying the calculus. In each case, we give both a mathematical definition of the combinator and the corresponding CHARITY code. There is a direct correspondence between these two levels. Some 'housekeeping' morphisms, like the *diagonal* $\triangle$, used in the former are more conveniently handled by the CHARITY term logic.

Interleaving. Although *interleaving*, a binary operator $|||\, : \nu \times \nu \longrightarrow \nu$, is not considered as a combinator in most process calculi, it is the simplest form of 'parallel' aggregation in the sense that it is independent of any particular interaction discipline. The definition below captures the intuition that the observations over the interleaving of two processes correspond to all possible interleavings of the observations of its arguments. Thus, one defines $|||\, = [\![\alpha_{|||}]\!]$, where

$$\begin{aligned}
\alpha_{|||} \;=\; \nu \times \nu &\xrightarrow{\triangle} (\nu \times \nu) \times (\nu \times \nu) \\
&\xrightarrow{(\omega \times \mathsf{id}) \times (\mathsf{id} \times \omega)} (\mathcal{P}(Act \times \nu) \times \nu) \times (\nu \times \mathcal{P}(Act \times \nu)) \\
&\xrightarrow{\tau_r \times \tau_l} \mathcal{P}(Act \times (\nu \times \nu)) \times \mathcal{P}(Act \times (\nu \times \nu)) \\
&\xrightarrow{\cup} \mathcal{P}(Act \times (\nu \times \nu))
\end{aligned}$$

The CHARITY code [2] for this

```
def bint: Pr(Ac(L)) * Pr(Ac(L))  -> Pr(Ac(L))
 = (t1, t2) =>
   (| (r1,r2) => bh: union(taur(bh r1, r2), taul(bh r2, r1))
    |) (t1,t2).
```

[2] Notice morphisms $\tau_r : \mathcal{P}(Act \times X) \times C \longrightarrow \mathcal{P}(Act \times (X \times C))$ and $\tau_l : C \times \mathcal{P}(Act \times X) \longrightarrow \mathcal{P}(Act \times (C \times X))$ stand for, respectively, the right and left *strength* associated to functor $\mathcal{P}(Act \times \mathsf{Id})$. They are straightforwardly encoded in CHARITY, *e.g.*, `def taur = (s,t) => set{(a,x) => (a, (x,t))} s.`

Restriction and Renaming. The *restriction* combinator $\backslash_K$, for each subset $K \subseteq L$, forbids the occurrence of actions in K. Formally, $\backslash_K = [\![\alpha_{\backslash_K}]\!]$ where

$$\alpha_{\backslash_K} \;=\; \nu \xrightarrow{\omega} \mathcal{P}(Act \times \nu) \xrightarrow{\mathsf{filter}_K} \mathcal{P}(Act \times \nu)$$

where $\mathsf{filter}_K = \lambda s\,.\;\; \{t \in s \mid \pi_1\, t \notin K\}$.

Once an interaction structure is fixed, any homomorphism $f : Act \longrightarrow Act$ lifts to a *renaming* combinator $[f]$ between processes defined as $[f] \;=\; [\![\alpha_{[f]}]\!]$, where

$$\alpha_{[f]} \;=\; \nu \xrightarrow{\omega} \mathcal{P}(Act \times \nu) \xrightarrow{\mathcal{P}(f \times \mathsf{id})} \mathcal{P}(Act \times \nu)$$

These two combinators are coded below as functions `bren` and `bret`, respectively. For convenience, the renaming homomorphism is represented as an endomorphism of `Ac(Lb(N))`.

```
def bren{eqL: Lb(N) * Lb(N) -> bool}:
     Pr(Ac(Lb(N))) * map(Ac(Lb(N)),Ac(Lb(N))) -> Pr(Ac(Lb(N)))

 = (t, h) =>
   (| r => bh: set{x =>
        { ff => x | ss a  => (a, p1 x)
        } app{eqA{eqL}}(compren h, p0 x) }  (bh r)
    |) t.


def bret{eqL: Lb(N) * Lb(N) -> bool}:
        Pr(Ac(Lb(N))) * set(Lb(N)) -> Pr(Ac(Lb(N)))

 = (t, k) =>
   (| r => bh:
       filter{x => not member{eqA{eqL}}(p0 x, compret k)}
       (bh r)
    |) t.
```

Any restriction set K of labels will have to be extended to a set of actions, by application of the embedding `act`, before it can be used in `bret`. Additionally, it may be 'completed' in order to cope with some syntactic conventions appearing in particular calculi. For example, to model Ccs, it becomes necessary to close K with respect to label complement (*i.e.*, the constructor `inv` in the Ccs label algebra implementation given above). Both tasks are achieved by function `compret` below, which should be tuned according to the syntactic particularities of the calculus under consideration. In the case of Ccs it will look like

```
def compret: set(Lb(N)) -> set(Ac(Lb(N)))
 = s => union( set{l => act(l)} s , set{l => act(inv(l)) } s).
```

Function `compren`, in the specification of `bren` does a similar completion of the renaming homomorphism.

Synchronous Product. This static operator models the simultaneous execution of its two arguments. At each step the resulting action is determined by the interaction structure for the calculus. Formally, $\otimes \;=\; [\![\alpha_\otimes]\!]$ where

$$\begin{aligned}\alpha_\otimes \;=\; \nu \times \nu &\xrightarrow{(\omega\times\omega)} \mathcal{P}(Act \times \nu) \times \mathcal{P}(Act \times \nu)\\ &\xrightarrow{\delta_r} \mathcal{P}(Act \times (\nu \times \nu)) \xrightarrow{\mathsf{sel}} \mathcal{P}(Act \times (\nu \times \nu))\end{aligned}$$

where $\mathsf{sel} = \mathsf{filter}_{\{0\}}$ filters out all synchronisation failures. Notice how interaction is catered by δ_r — the distributive law for the strong monad $\mathcal{P}(Act \times \mathsf{Id})$. In fact, the monoidal structure in Act extends functor $\mathcal{P}(Act \times \mathsf{Id})$ to a strong monad, δ_r being the Kleisli composition of the left and the right strengths. This, on its turn, involves the application of the monad multiplication to 'flatten' the result and this, for a monoid monad, requires the suitable application of the underlying monoidal operation. This, in our case, fixes the interaction discipline. Going pointwise:

$$\delta_r^{\mathcal{P}(Act\times\mathsf{Id})}\ \langle c_1, c_2\rangle \;=\; \{\langle a'\theta a, \langle p, p'\rangle\rangle |\ \langle a, p\rangle \in c_1 \ \wedge\ \langle a', p'\rangle \in c_2\}$$

In Charity,

```
def bsyn{eqL: Lb(N) * Lb(N) -> bool}:
        Pr(Ac(Lb(N))) * Pr(Ac(Lb(N))) -> Pr(Ac(Lb(N)))

 = (t1, t2) =>
      (| (r1,r2) => bh:  sel{eqL} deltar{eqL} (bh r1, bh r2) |) (t1,t2).
```

where `deltar` and `sel` implement morphisms δ_r and sel, respectively.

Parallel Composition. Parallel composition arises as a combination of *interleaving* and *synchronous product*, in the sense that the evolution of $p \mid q$, for processes p and q, consists of all possible derivations of p and q plus the ones associated to the synchronisations allowed by the particular interaction structure for the calculus. This cannot be achieved by mere composition of the corresponding combinators $|||$ and $\otimes$: it has to be performed at the 'genes' level for $|||$ and $\otimes$. Formally, $\mid \;=\; [\![\alpha_\mid]\!]$, where

$$\begin{aligned}\alpha_\mid \;=\; \nu \times \nu &\xrightarrow{\Delta} (\nu \times \nu) \times (\nu \times \nu)\\ &\xrightarrow{(\alpha_{|||}\times\alpha_\otimes)} \mathcal{P}(Act \times (\nu \times \nu)) \times \mathcal{P}(Act \times (\nu \times \nu))\\ &\xrightarrow{\cup} \mathcal{P}(Act \times (\nu \times \nu))\end{aligned}$$

which is coded in Charity as

```
def bpar{eqL: Lb(N) * Lb(N) -> bool}:
        Pr(Ac(Lb(N))) * Pr(Ac(Lb(N))) -> Pr(Ac(Lb(N)))

 = (t1, t2) =>
   (| (r1,r2) => bh: union(
                  sel{eqL} deltar{eqL} (bh r1, bh r2),
                  union(taur(bh r1, r2), taul(bh r2, r1)))
    |) (t1,t2).
```

4 A Process Language

There is so far no place for *recursive* processes in the approach we have been discussing. The possibility of supplying the dynamics of each particular example as a particular 'gene' coalgebra, is unsatisfactory in this respect. Instead, the obvious way to deal with *recursive* processes in general consists of defining a *language* whose terms stand for process expressions, including a construction involving process *variables* as valid terms. Such variables should be bound, in whatever we understand as the interpreter *environment*, by process equations of the form $v = exp$, where v is a variable and exp a process expression. We have, however, to proceed with some care as it is well-known that not all defining equations determine process behaviour in an unique way. For example, not only any process is a solution to $v = v$, but also equations like $v = v|v$ admit different, no bisimilar, solutions. One way of ensuring the existence of unique solutions, is to require that all variables occurring on the right hand side of an equation are guarded, in the sense that they are bounded by a prefix combinator. This has been proved in [21] as a sound criteria for the existence of unique solutions to what is called there *strict bisimulation equations* which, recall, correspond to equality in the final coalgebra. In fact, in [21], guardedness is only required for variables wrt expressions in which they occurr. The extension, assumed here, of this requirement to all variables in an expression facilitates the development of the interpreter and does not appear to be a major restriction. Therefore, our process language will include a prefix-like construction — `pvar` — to introduce (guarded) variables in an expression.

Summing up, we are left with the tasks of defining a term language for processes, its interpretation in the (final) semantic model and a suitable model for the environment collecting the relevant process defining equations. Let us tackle them, one at a time.

The Language. As expected, a term language for processes, over a set `L` of labels, will be defined as an inductive type. The Charity declaration below introduces `Ln(L)` as the initial algebra for a functor Σ induced by the following BNF description:

$$
\begin{array}{lll}
\langle \mathtt{P} \rangle ::= \mathtt{pnil} & \mid \mathtt{ppre}(a, \langle \mathtt{P} \rangle) & \mid \mathtt{pcho}(\langle \mathtt{P} \rangle, \langle \mathtt{P} \rangle) \mid \\
\quad\quad\quad \mathtt{pint}(\langle \mathtt{P} \rangle, \langle \mathtt{P} \rangle) & \mid \mathtt{psyn}(\langle \mathtt{P} \rangle, \langle \mathtt{P} \rangle) & \mid \mathtt{ppar}(\langle \mathtt{P} \rangle, \langle \mathtt{P} \rangle) \mid \\
\quad\quad\quad \mathtt{pret}(\langle \mathtt{P} \rangle, K) & \mid \mathtt{pren}(\langle \mathtt{P} \rangle, f) & \mid \mathtt{pvar}(a, i)
\end{array}
$$

where $a \in \mathtt{Ac(L)}$, $K \subseteq \mathtt{L}$, i is a process variable and f a renaming $\mathtt{Ac(L)}$ homomorphism. Constructors `pnil`, `ppre`, `pcho`, `pint`, `psyn`, `ppar`, `pret` and `pren` correspond to the different process combinators. The only exception is `pvar`, which builds a new process given an action and a process variable. Its semantics is defined similarly to the one of the prefix combinator, according to the discussion above.

The equivalent CHARITY declaration follows. Note the definition is sufficiently generic, as it is parametric on `L` and resorts to whatever interaction structure is provided for `Ac(L)`:

```
data Ln(L) -> P =
     pnil: 1 -> P | ppre: Ac(L) * P -> P |
     pcho: P * P -> P | pint: P * P -> P |
     psyn: P * P -> P | ppar: P * P -> P |
     pret: P * set(L) -> P |
     pren : P * map(Ac(L), Ac(L)) -> P |
     pvar: Ac(L) * string -> P.
```

The Interpreter. How can `Ln(L)` expressions be interpreted as processes? Within the initial algebra approach to semantics, once the syntax is fixed, a semantic Σ-algebra would be produced and the interpretation defined as the associated catamorphism. Our semantic universe, however, is the final coalgebra for functor $\mathcal{P}(\mathtt{Ac(L)} \times \mathsf{Id})$, and, therefore, the dual final semantics approach will be followed up. What has to be done is to cast the syntax, *i.e.*, the set of terms $\mathtt{Ln(L)}$, into a $\mathcal{P}(\mathtt{Ac(L)} \times \mathsf{Id})$-coalgebra. The interpreter will follow as the associated *anamorphism.*

Let function $\mathtt{sem} : \mathtt{Ln(L)} \longrightarrow \mathtt{Pr(Ac(L))}$ stand for the desired interpreter. The 'gene' for this anamorphism is a 'syntactic' coalgebra $\mathtt{syn} : \mathtt{Ln(L)} \longrightarrow \mathcal{P}(\mathtt{Ac(L)} \times \mathtt{Ln(L)})$ which computes the 'syntactical' derivations of process terms. Observe, now, that the 'canonical' way to define `syn` is as a Σ-catamorphism. Its 'gene' is denoted by $\alpha_{\mathtt{syn}}$ in the sequel. The diagram below contains the 'full' picture, where $\alpha_{\mathtt{syn}}$ is actually the *only* function to be supplied by the programmer:

$$\begin{tikzcd}
\mathtt{Pr(Ac(L))} \arrow[r, "\omega"] & \mathcal{P}(\mathtt{Ac(L)} \times \mathtt{Pr(Ac(L))}) \\
\mathtt{Ln(L)} \arrow[u, "\mathtt{sem}"] \arrow[r, "\mathtt{syn}"] & \mathcal{P}(\mathtt{Ac(L)} \times \mathtt{Ln(L)}) \arrow[u, "\mathcal{P}(\mathsf{id} \times \mathtt{sem})"'] \\
\Sigma\, \mathtt{Ln(L)} \arrow[u, "\alpha"] \arrow[r, "\Sigma\, \mathtt{syn}"] & \Sigma\, \mathcal{P}(\mathtt{Ac(L)} \times \mathtt{Ln(L)}) \arrow[u, "\alpha_{\mathtt{syn}}"']
\end{tikzcd}$$

Then, we get for free

$$\mathtt{syn} \;=\; (\!|\alpha_{\mathtt{syn}}|\!)_{\Sigma}$$

and

$$\mathtt{sem} \;=\; [\!(\mathtt{syn})\!]_{\mathcal{P}(\mathtt{Ac(L)} \times \mathsf{Id})}$$

where ω and α are, respectively, the final $\mathcal{P}(\mathtt{Ac(L)} \times \mathsf{Id})$-coalgebra and the initial Σ-algebra.

Environment. Our last concern is the introduction of an environment to the interpreter in order to collect all the process defining equations relevant to conduct experiments on a particular network of processes. Such an environment E can be thought of as mapping assigning to each process variable an expression in `Ln(L)`. Assuming variable identifiers are modeled by strings, E will be typed in CHARITY as `map(string, Ln(L))`. Clearly, E acts as a supplier of context information to the interpretation function `sem`, whose type is

$$\mathtt{sem} : \mathtt{Ln(L)} \times E \longrightarrow \mathtt{Pr(Ac(L))}$$

All types in CHARITY are *strong* and, therefore, this extra parameter is smoothly accommodated in our framework. In fact, both `sem` and `syn` become defined as, respectively, a *strong anamorphism* for $\mathcal{P}(\mathtt{Ac(L)} \times \mathsf{Id})$ and a *strong catamorphism* for Σ. The diagram above remains valid, but it has to be interpreted in the Kleisli category for the product comonad. Its interpretation in the original category is depicted in the diagram below, which makes explicit the structure involved by including dotted lines to correctly type an arrow in the Kleisli as an arrow in the original category. Symbols ω' and α' denote, respectively, the corresponding final coalgebra and initial algebra in the Kleisli, defined simply as $\omega \cdot \pi_1$ and $\alpha \cdot \pi_1$ (*cf.*, proposition 4.3 in [7]). Thus,

$$\begin{tikzcd}
\mathtt{Pr(Ac(L))} \times E \arrow[rr, "\omega'"] \arrow[d, dotted, no head] & & \mathcal{P}(\mathtt{Ac(L)} \times \mathtt{Pr(Ac(L))}) \\
\mathtt{Pr(Ac(L))} & & \\
\mathtt{Ln(L)} \times E \arrow[u, "\mathtt{sem}"] \arrow[rr, "\mathtt{syn}"] \arrow[d, dotted, no head] & & \mathcal{P}(\mathtt{Ac(L)} \times \mathtt{Ln(L)}) \arrow[r, dotted, no head] & \mathcal{P}(\mathtt{Ac(L)} \times \mathtt{Ln(L)}) \times E \arrow[uu] \\
\mathtt{Ln(L)} & & & \\
\Sigma\, \mathtt{Ln(L)} \times E \arrow[u, "\alpha'"] \arrow[rr] & & \Sigma\, \mathcal{P}(\mathtt{Ac(L)} \times \mathtt{Ln(L)}) \arrow[r, dotted, no head] \arrow[uu, "\alpha_{\mathtt{syn}}"] & \Sigma\, \mathcal{P}(\mathtt{Ac(L)} \times \mathtt{Ln(L)}) \times E
\end{tikzcd}$$

Fig. 1. The overall interpreter diagram

Formally, the interpretation function arises as

$$\mathtt{sem} \;=\; \mathsf{unfold}_{\mathcal{P}(\mathtt{Ac(L)} \times \mathsf{Id})}\, \mathtt{syn}$$

where

$$\mathtt{syn} \;=\; \mathsf{fold}_{\Sigma}\, \alpha_{\mathtt{syn}}$$

and unfold /fold stand, respectively, for the strong anamorphism and catamorphism combinator.

The actual picture is slightly more complex, however, as the definition of `syn` is made in terms of both the computations on the substructures of its argument and these substructures themselves. To be precise, it arises as a *strong paramorphism*. *Paramorphisms* [19] generalise catamorphisms in order to deal with cases in which the result of a recursive function depends not only on computations in substructures of its argument, but also on the substructures themselves. The recursion pattern it entails, in the particular case of $\mathbb{N}$, is known as *primitive recursion*.

In the general case, a *paramorphism* is defined as the unique arrow making the following diagram to commute.

$$\begin{tikzcd}
X & \mathsf{T}\,(X \times \mu_{\mathsf{T}}) \arrow[l, "f"'] \\
\mu_{\mathsf{T}} \arrow[u, "\mathsf{par}_{\mathsf{T}}\, f"] & \mathsf{T}\,\mu_{\mathsf{T}} \arrow[l, "\alpha_{\mathsf{T}}"] \arrow[u, "\mathsf{T}\,\langle \mathsf{par}_{\mathsf{T}}\, f, \mathsf{id}\rangle"']
\end{tikzcd}$$

Notice the domain of the 'target algebra' is now the T image of its carrier with the inductive type itself. Paramorphisms have no direct implementation in CHARITY, but we can program their transformation to catamorphisms captured by law

$$\mathsf{par}_{\mathsf{T}}\, f \;=\; \pi_1 \cdot (\!|\langle f, \alpha_{\mathsf{T}} \cdot \mathsf{T}\,\pi_2\rangle|\!)_{\mathsf{T}}$$

The language, however, provides an easier way of dealing with functions defined as paramorphisms, by using '`#`'s. Inside a fold, `#` stands for the value being currently analysed, *before* the recursive application. This is exactly what is done in the interpreter `sem` presented in Appendix A. Not much remains to be said about this definition as the encoding of each process combinator has already been detailed in the previous section. Just note the interpretation of the new construction `pvar(a,i)` is as expected: the continuation process arises as the interpretation of the process expression associated to variable `i`, if `i` is collected in the environment, or `pnil` otherwise.

To fully understand the definition, observe that the derivations of a process expression are (a set of pairs of actions and) *process expressions*, whereas, in the previous definition of 'stand-alone' combinators they were defined in terms of *processes* themselves. As an illustration, compare the entry corresponding to renaming in the 'gene' of `syn` with the definition of `bren` in the previous section. This same observation justifies the auxiliary definitions of `stau1`, `stau2`, `ssel`, `sdelta1` and `sdelta2`, whose role is similar to the original `taur`, `sel` and `deltar`.

5 Conclusions

This paper specifies process combinators in a way which is directly translatable into the CHARITY programming language. In this way, a functional implementation of a (family of) of calculi becomes available in which experiments can be carried out. By experiments one means that a process expression is supplied to the system and its evolution traced. In fact, all the allowed derivations are computed step by step, resorting to the CHARITY evaluation mechanism for coinductive types. Experimenting with process prototypes is not essentially different from animating data oriented specifications in any of the rapid prototyping systems popular among the formal methods community. The difference lies in the underlying shift towards coinduction, namely final semantics for processes, an active research area triggered by Aczel's landmark paper [1]. Other recent contributions include, for example, references [28] and [18] on model theory for CSP and π-calculus, respectively. Our approach is certainly more programming oriented. Rather than foundational, it favours both *calculation* [3] and animation. Current work includes the full development of a prototyping kernel for this kind of calculi specifications, incorporating an interface layer to provide a more adequate user interface for prototype testing. We feel challenged by *mobility* [22], whose extension in this framework should incorporate variable binding and dynamically generated names and localities.

Although we have illustrated our approach to process prototyping going through, in some detail, a particularly well known family of such calculi in the CCS tradition, we would like to stress its *genericity* at three distinct levels.

First of all, the *behaviour model* is a parameter of the calculus (and of its interpreter). Recall that processes were defined as inhabitants of the carrier of the final coalgebra for $\mathsf{T} = \mathsf{B}\ (Act \times \mathsf{Id})$, for B the finite powerset functor. We have shown that, assuming a commutative monoidal structure over Act, the behaviour model captured by $\mathcal{P}(Act \times \mathsf{Id})$ is a *strong Abelian monad*. The proposed constructions remain valid for any instantiation of B by such a monad. For example, taking B the identity monad ($\mathsf{B} = \mathsf{Id}$), leads to a calculus of *deterministic* (and perpetual) processes. A further elaboration of this would replace Id by $\mathsf{Id} + \mathbf{1}$, entailing a calculus for deterministic, but *partial* processes in the sense that the derivation of a 'dead' state is always possible. Such a calculus is far less expressive than the one dealt with in this paper. In fact, derivations do not form any kind of *collection* and, thus, non-determinism is ruled out. Therefore, combinators which explore non-determinism, lack a counterpart here. Such is the case of *choice*, *interleaving* and *parallel*. On the other hand, the composition of $\mathsf{Id} + \mathbf{1}$ with the monoidal monad generated by Act is still a strong Abelian monad, and therefore *synchronous product* is still definable. A limited form of non-determinism is recovered, however, by taking B as the sequence monad. Families of calculi of partial and ordered processes would be declared in CHARITY as, respectively,

```
data C -> PartialPr(A) = bh: C -> SF(A * C).
```

and

```
data C -> OrderedPr(A) = bh: C -> list(A * C).
```

A second source of genericity, orthogonal to the one above, is the separate specification of an *interaction structure*. Note that all the process combinators introduced in this paper are either *independent* of any particular interaction discipline or *parametrized* by it.

Finally, we are by no means limited to functors of the $\mathsf{T} = \mathsf{B}\ (Act \times \mathsf{Id})$ family, which, as mentioned above, correspond to an 'active' interpretation of the process structure. For example, an universe for reactive processes may be specified as a final coalgebra for $\mathsf{T}\ X = (\mathsf{B}\ X)^{Act}$, parametrized, again, on a behaviour monad and an interaction structure.. Taking B as the finite powerset monad, such processes are declared as

```
data C -> ReactivePr(A) = bh: C -> A => set(C).
```

This kind of *genericity* is the main concern of related research on *state-based componentware* is reported in [4].

Acknowledgements. The authors wish to thank the *Logic and Formal Methods* group at Minho for the smooth working environment they had the chance to benefit from. Comments by Jan Rutten and Luis Monteiro about research described in this paper are gratefully acknowledged.

References

1. P. Aczel. Final universes of processes. In B. et al, editor, *Proc. Math. Foundations of Programming Semantics*. Springer Lect. Notes Comp. Sci. (802), 1993.
2. L. S. Barbosa. *Components as Coalgebras*. PhD thesis, DI, Universidade do Minho, 2001.
3. L. S. Barbosa. Process calculi *à la* Bird-Meertens. In *CMCS'01*, volume 44.4, pages 47–66, Genova, April 2001. Elect. Notes in Theor. Comp. Sci., Elsevier.
4. L. S. Barbosa and J. N. Oliveira. State-based components made generic. In H. P. Gumm, editor, *CMCS'03, Elect. Notes in Theor. Comp. Sci.*, volume 82.1. Elsevier, 2003.
5. R. Bird and O. Moor. *The Algebra of Programming*. Series in Computer Science. Prentice-Hall International, 1997.
6. R. Cockett and T. Fukushima. About Charity. Yellow Series Report No. 92/480/18, Dep. Computer Science, University of Calgary, June 1992.
7. R. Cockett and D. Spencer. Strong categorical datatypes I. In R. A. G. Seely, editor, *Proceedings of Int. Summer Category Theory Meeting, Montréal, Québec, 23–30 June 1991*, pages 141–169. AMS, CMS Conf. Proceedings 13, 1992.
8. R. Cockett and D. Spencer. Strong categorical datatypes II: A term logic for categorical programming. *Theor. Comp. Sci.*, 139:69–113, 1995.
9. T. Hagino. A typed lambda calculus with categorical type constructors. In D. H. Pitt, A. Poigné, and D. E. Rydeheard, editors, *Category Theory and Computer Science*, pages 140–157. Springer Lect. Notes Comp. Sci. (283), 1987.

10. P. Henderson. me too: A language for software specification and model building. Preliminary Report, University of Stirling, 1984.
11. B. Jacobs. Objects and classes, co-algebraically. In C. L. B. Freitag, C.B. Jones and H.-J. Schek, editors, *Object-Orientation with Parallelism and Persistence*, pages 83–103. Kluwer Academic Publishers, 1996.
12. B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–159, 1997.
13. P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997.
14. B. Jay and J. Cockett. Shaply types and shape polymorphism. In D. Sannella, editor, *Programming Languages and Systems — ESOP'94*, pages 302–316. Springer Lect. Notes Comp. Sci. (788), 1994.
15. C. B. Jones. Specification and design of (parallel) programs. In R. E. A. M. (IFIP), editor, *Information Processing 83*, pages 321–332. Elsevier Science Publishers B. V. (North-Holland), 1983.
16. R. B. Kieburtz. Codata and comonads in Haskell. Unpublished manuscript, 1998.
17. A. Kock. Strong functors and monoidal monads. *Archiv für Mathematik*, 23:113–120, 1972.
18. M. Lenisa. *Themes in Final Semantics*. PhD thesis, Universita de Pisa-Udine, 1998.
19. L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–425, 1992.
20. E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proceedings of the 1991 ACM Conference on Functional Programming Languages and Computer Architecture*, pages 124–144. Springer Lect. Notes Comp. Sci. (523), 1991.
21. R. Milner. *Communication and Concurrency*. Series in Computer Science. Prentice-Hall International, 1989.
22. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (parts I and II). *Information and Computation*, 100(1):1–77, 1992.
23. C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Technische Hochschule Darmstadt, 1962.
24. J. Rutten. Universal coalgebra: A theory of systems. *Theor. Comp. Sci.*, 249(1):3–80, 2000. (Revised version of CWI Techn. Rep. CS-R9652, 1996).
25. T. Sheard. Type parametric programming. Technical report, Oregon Graduate Institute of Science and Technology, Portland, USA, 1993.
26. D. Turi and J. Rutten. On the foundations of final coalgebra semantics: non-well-founded sets, partial orders, metric spaces. *Math. Struct. in Comp. Sci.*, 8(5):481–540, 1998.
27. G. Winskel and M. Nielsen. Models for concurrency. In S. Abramsky, D. M. Gabbay, and T. S. E. Gabbay, editors, *Handbook of Logic in Computer Science (vol. 4)*, pages 1–148. Oxford Science Publications, 1995.
28. U. Wolter. A coalgebraic introduction to csp. In *CMCS'99, Elect. Notes in Theor. Comp. Sci.*, volume 19. Elsevier, 1999.
29. G. C. Wraith. A note on categorical data types. In D. e. a. Pitt, editor, *Proc. Category Theory and Computer Science*, pages 118–127. Springer Lect. Notes Comp. Sci. (389), 1988.

A The Interpreter

```
def sem{eqL: Lb(N) * Lb(N) -> bool}:
     Ln(Lb(N)) * map(string, Ln(Lb(N)))  -> Pr(Ac(Lb(N)))

    = (exp,m)  => (| e => bh: syn{eqL}(e,m) |) exp.


def syn{eqL: Lb(N) * Lb(N) -> bool}:
     Ln(Lb(N)) * map(string, Ln(Lb(N)))
     ->  set(Ac(Lb(N)) * Ln(Lb(N)))

 = (pr,m) =>
   {| pnil: ()      => empty
    | ppre: (a,l)   => sing(a, p1 #)
    | pvar: (a,s)   => { ss(p) => sing(a, p)
                       | ff    => empty
                       } app{eq_string}(m,s)
    | pcho: (lp,lq) => union(lp, lq)
    | pint: (lp,lq) => union(stau1(lp, p1 #), stau1(lq,  p0 #))
    | psyn: (lp,lq) => ssel{eqL} sdelta1{eqL} (lp,lq)
    | ppar: (lp,lq) => union(union(stau2(lp, p1 #), stau2(lq,  p0 #)),
                             ssel{eqL} sdelta2{eqL} (lp,lq))
    | pret: (l,k)   => set{x => (p0 x, pret(p1 x, k))}
               filter{x => not member{eqA{eqL}}(p0 x, compren k)} l
    | pren: (l,h)   => set{x => { ff    => (p0 x,  pren(p1 x, h))
                                | ss a  => (a, pren(p1 x, h))
                                } app{eqA{eqL}}(compret h, p0 x) } l
    |} pr.
```

where

```
def stau1: set(Ac(L) * Ln(L)) * Ln(L) -> set(Ac(L) * Ln(L))
    = (s,p) => set{(a,x) => (a, pint(x,p))} s.

def ssel{eql: L * L -> bool}: set(Ac(L) * B)  -> set(Ac(L) * B)
    = s => filter{x => not eqA{eql}(p0 x, nop)} s.
```

B Programming in Charity

Charity [6] makes only a few assumptions on the underlying semantics category: it assumes distributivity and replaces the Cartesian closedness requirement, implicit in the original approaches to categorical types, namely in the [29] refinement of Hagino's work, by the assumption that all datatypes are *strong*. Charity primitive types are, then, the nullary (denoted by `1`) and binary product types (denoted by the infix operator `*`, with projections `p0` and `p1`). The absence of exponentials at the basic level of the language gives to programming in Charity a rather different flavour when compared with more traditional functional languages. In particular, functions are not values and function composition, instead of function application, is taken as the fundamental primitive in the language. This does not mean, however, that Charity lacks support for higher-order types: simply they have to be explicitly declared.

In this context, Charity may be classified as a polymorphic, strongly-typed language which is functional in style. In particular, any program has a guarantee of 'termination', in the sense that the term representing it always reduces to a head normal form and, therefore, a 'response' is produced. Such a 'response' is computed either lazily or eagerly depending on the types involved being coinductive or inductive, respectively. In any case, the type system simply blocks the possibility of writing functions that may never terminate. Although both data and programs can be expressed pointfree in terms of such categorical combinators, programming at such a level becomes rather awkward (namely, by the number of projections often needed to distribute variables along an expression). Charity programs are written in a term logic [8] for distributive categories enriched with a definitional mechanism for inductive and coinductive strong datatypes. This allows the use of variables in combinator expressions and pattern matching.

Let us briefly review the main 'building blocks' of a Charity program, starting with type declarations. The declaration of a coinductive type in Charity has the following format:

$$\texttt{data}\ S\texttt{->}T\texttt{(}A\texttt{)}\ \texttt{=}\ o_1\texttt{:}S\texttt{->}E_1(A,S)\ \texttt{|}\ \ldots\ \texttt{|}\ o_n\texttt{:}S\texttt{->}E_n(A,S)\texttt{.}$$

which introduces the type $T\texttt{(}A\texttt{)}$, parametric on A. The declaration format conveys the idea that morphisms from any type S to $T\texttt{(}A\texttt{)}$ are solely determined by morphisms from S to each $E_i(A,S)$, the output type of observer o_i. Formally, this defines $T\texttt{(}A\texttt{)}$ as the final coalgebra for a functor T_A determined by the observers signature, *i.e.*,

$$\langle \nu_{\mathsf{T}_\mathsf{A}}, \langle o_1, \ldots o_n\rangle : \nu_{\mathsf{T}_\mathsf{A}} \longrightarrow \prod_i E_i(A, \nu_{\mathsf{T}_\mathsf{A}})\rangle$$

Each o_i identifies one of such observers whose type is obtained by setting $S = T\texttt{(}A\texttt{)}$ in the declaration. Therefore, $T\texttt{(}A\texttt{)}$ models $\nu_{\mathsf{T}_\mathsf{A}}$.

Dually, an inductive type is declared as

$$\texttt{data}\ T\texttt{(}A\texttt{)}\texttt{->}S\ \texttt{=}\ c_1\texttt{:}E_1(A,S)\texttt{->}S\ \texttt{|}\ \ldots\ \texttt{|}\ c_n\texttt{:}E_n(A,S)\texttt{->}S\texttt{.}$$

Such a declaration introduces type $T\texttt{(}A\texttt{)}$, again parametric on A, as the initial algebra

$$\langle \mu_{\mathsf{T}_\mathsf{A}}, [c_1, \ldots, c_n] : \sum_i E_i(A, \mu_{\mathsf{T}_\mathsf{A}}) \longrightarrow \mu_{\mathsf{T}_\mathsf{A}}\rangle$$

The basic combinator associated to a coinductive type is `unfold`, *i.e.* an anamorphism in a strong setting. In Charity, this is specified by supplying, for each observer

o_i the corresponding component p_i of the source coalgebra. As expected, strongness requires that each p_i be typed as $p_i : S \times C \longrightarrow E_i(A, S)$, assuming S as the carrier of the source coalgebra and C the context type. The concrete syntax for an `unfold` expression is as follows:

$$\texttt{(}s\texttt{,}c\texttt{) => (| } s \texttt{=>} o_1\texttt{:}p_1\texttt{(}s\texttt{,}c\texttt{) | } \cdots \texttt{ | } o_n\texttt{:}p_n\texttt{(}s\texttt{,}c\texttt{) |)}$$

where s and c denote variables of type S and C, respectively. A 'degenerate', *i.e.*, non recursive, `unfold`, is the `record` combinator which provides a way of populating a coinductive type by specifying particular values for the observers. The general pattern for the `record` combinator is

$$c \texttt{=> (} o_1\texttt{:}f_1\texttt{(}c\texttt{)} \cdots \texttt{ | } o_n\texttt{:}f_n\texttt{(}c\texttt{)} \texttt{)}$$

For inductive types the duals of these two combinators are `fold` and `case`, respectively. The `fold` combinator is specified by introducing, for each constructor c_i, the corresponding constructor, d_i, of the target algebra. Each of them is typed as $d_i : E_i(A, S) \times C \longrightarrow S$, where C is the type of the context and S the carrier of the target algebra. The target algebra is, of course, just 'the *either* of all such d_i'. The concrete syntax for a `fold` expression is:

$$\texttt{(}s\texttt{,}c\texttt{) => \{| } c_1\texttt{:}s_1 \texttt{=>} d_1\texttt{(}s_1\texttt{,}c\texttt{) | } \cdots \texttt{ | } c_n\texttt{:}s_n \texttt{=>} d_n\texttt{(}s_n\texttt{,}c\texttt{) |\}} \; s$$

As mentioned above, the `record` combinator provides a canonical way of specifying (generalized) elements of a coinductive type. Dually, (generalized) elements of any type S, having an inductive type as domain of variation, can arise in a simple (non recursive) way by defining its value on each constructor of the domain. This construction is known in Charity as the `case` combinator whose syntax, in the general case, is

$$\texttt{(}s\texttt{,}c\texttt{) => \{ } c_1 s_1 \texttt{=>} d_1\texttt{(}s_1\texttt{,}c\texttt{) | } \cdots \texttt{ | } c_n s_n \texttt{=>} \qquad d_n\texttt{(}s_n\texttt{,}c\texttt{) \}} \; s$$

Each d_i is a function from $E_i(A, \mu_{\mathsf{T_A}}) \times C$, to the target type. Notice that $E_i(A, \mu_{\mathsf{T_A}})$ is the domain of constructor c_i and C denotes the type of the context. Therefore, the domain of the `case` combinator is $\mathsf{T_A} \times C$. Using strength, context is pushed inside the $\mathsf{T_A}$ outermost (coproduct) structure and, therefore, even in this general case, the combinator is still determined by $[d_1, \ldots, d_n]$.

For each inductive or coinductive type, the `map` combinator denotes the action on morphisms, with strength, of the corresponding type functor. Its general format, for $h_i : A_i \times C \longrightarrow A'_i$, is:

$$\texttt{(}t\texttt{,}c\texttt{) => } T \texttt{\{ } x_1 \texttt{=>} h_1\texttt{(}x_1\texttt{,}c\texttt{) | } \cdots \texttt{ | } x_m \texttt{=>} h_m\texttt{(}x_m\texttt{,}c\texttt{) \}} \; t$$